64000

HP 64000 Logic Development System

Model 64822AF/AT C/64000 Compiler Supplement 6809 Family

CERTIFICATION

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

WARRANTY

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country.

HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

LIMITATION OF WARRANTY

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

EXCLUSIVE REMEDIES

THE REMEDIES PROVIDED HEREIN ARE BUYER'S SOLE AND EXCLUSIVE REMEDIES. HP SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY.

ASSISTANCE

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.

FOLD HERE

HEWLETT PACKARD

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 1303 COLORADO SPRINGS, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

HEWLETT-PACKARD
Logic Product Support Dept.
Attn: Technical Publications Manager
Centennial Annex - D2
P.O. Box 617
Colorado Springs, Colorado 80901-0617

FOLD HERE

Your cooperation in completing and returning this form will be greatly appreciated. Thank you.

READER COMMENT SHEET

Operating Manual, Model 64822AF/AT
C/64000 Compiler Supplement for 6809/6809E
64822-90901, May 1985

Your comments are important to us. Please answer this questionaire and return it to us. Circle the number that best describes your answer in questions 1 through 7. Thank you.

1. The information in this book is complete:

Doesn't cover enough (what more do you need?)	1 2 3 4 5	Covers everything

2. The information in this book is accurate:

Too many errors	1 2 3 4 5	Exactly right

3. The information in this book is easy to find:

I can't find things I need	1 2 3 4 5	I can find info quickly

4. The !ndex and Table of Contents are useful:

Helpful	1 2 3 4 5	Missing or inadequate

5. What about the "how-to" procedures and examples:

No help	1 2 3 4 5	Very helpful
Too many now	1 2 3 4 5	I'd like more

6. What about the writing style:

Confusing	1 2 3 4 5	Clear

7. What about organization of the book:

Poor order	1 2 3 4 5	Good order

8. What about the size of the book:

too big/small	1 2 3 4 5	Right size

Comments: _____

_____

_____

_____

Particular pages with errors?

_____

Name (optional): _____
Job title: _____
Company: _____
Address: _____

Note: If mailed outside U.S.A., place card in envelope. Use address shown on other side of this card.

HEWLETT PACKARD

OPERATING MANUAL

MODEL 64822AF/AT
C/64000 COMPILER SUPPLEMENT
FOR 6809/6809E

Manual Part No. 64822-90901

PRINTED: MAY 1985

Printing History

Each new edition of this manual incorporates all material updated since the previous edition. Manual change sheets are issued between editions, allowing you to correct or insert information in the current edition.

The part number on the back cover changes only when each new edition is published. Minor corrections or additions may be made as the manual is reprinted between editions. Vertical bars in a page margin indicate the location of reprint corrections.

First Printing ..... March 1983 (P/N 64822-90901)
Second Printing..... May 1985 (P/N 64822-90901)

Software Version Number

Your HP 64000 software is identified with a version number in the form XX.YY. The version number is printed on a label attached to the software media or media envelope. This manual applies to the following:

Model HP 64822AF Version 1.YY

Within the software version number, the digit to the left of the decimal point indicates the product feature set. This manual supports all software versions identified with this same digit.

The digits to the right of the decimal point indicate feature subsets. These feature subsets normally have no affect on the manual. However, if you subscribe to the "Software Material Subscription" (SMS), these subset items are covered in the "Software Response Bulletin" (SRB).

Software Materials Subscription

Hewlett-Packard offers a Software Materials Subscription (SMS) to provide you with the most timely and comprehensive information concerning your HP 64000 Logic Development System. This service can maximize the productivity of your HP system by ensuring that you have the latest product enhancements, software revisions, and software reference manuals.

For a more detailed description of the SMS, refer to chapter 1.

Duplicating Software

Before using the flexible disc(s) provided with this product, make a work copy. Retain the original disc(s) as the master copy and use the work copy for daily use.

Specific rights to use one copy of the software product(s) are granted for use on a single, standalone development station or a cluster of development stations which boot from a single mass storage device.

Should your master copy become lost or damaged, replacement discs are available through your Hewlett-Packard sales and service office.

Table of Contents

Table of Contents (Cont'd)

Chapter 3: Run-time Library Specifications (Cont'd)

Chapter 4: Run-time Library Specifications for Real Numbers

Appendix A: Run-Time Error Descriptions

List of Tables

NOTES

Chapter 1
C/64000 COMPILER 6809

INTRODUCTION

This compiler supplement is an extension of the C/64000 Compiler Reference Manual. It contains all the processor-dependent compiler information for use with the 6809 microprocessor.

Descriptions of compiler features, options, and their use are supplied. A detailed discussion of the run-time libraries required by the 6809 code generator is included. In addition, a brief discussion of the features, capabilities, and limitations of C program development using the emulator is provided.

C PROGRAM DESIGN

C programs should be designed to be as processor and implementation independent as possible, yet certain concessions must be made when the processor has unique characteristics. Programs written to run on a large mainframe computer with megabytes of virtual memory may not run on a 6809 with a maximum of 64k-bytes of addressable memory. Most large mainframe computer implementations have enough memory to allocate a stack area and a heap for dynamic memory allocation with no prompting by the user. In a limited memory system these factors must be communicated to the compiler in some manner. For the 6809, the user must specify the location of the stack and, if needed, the location of a memory pool for dynamic allocation routines. The following sections describe subjects related to programming and compiling C/64000 for the 6809 processor.

HOW TO IMPLEMENT A PROGRAM

The usual process of software generation is as follows:

a. Create a source program file using the editor.

b. Compile the source program.

c. Link the relocatable files.

d. Emulate the absolute file.

e. Debug as necessary.

This chapter will provide insight into each of these processes.

The Source File

The C/64000 compiler takes as input a program source file created with the editor. The basic form of a source file is:

```
"C"
"6809"
/* C functions */

function_name (argument list, if any)
argument declarations, if any
{
   declarations
   statements
}
   .
   .
/*C PROGRAM */
main (argument list, if any)
main argument declarations, if any
{
   main declarations
   main statements
}
   .
   .
```

When source file editing is complete, it is ready for compilation. Notice that the first line of source code contains the special compiler directive "C". This informs the compiler to use the C language subsystem. The second line contains the special compiler directive "6809". This directive informs the compiler to generate relocatable code for the 6809 microprocessor.

The C/64000 compiler can use either all upper-case characters for keywords or all lower-case characters for keywords. The option $UPPER_KEYS OFF$ (default condition) allows lower-case keywords to be recognized.

The compiler may produce up to four files as output: a relocatable file, a listing file (if specified), an assembly file (if specified), and an assembly symbol file (if specified). Descriptions of these files are as follows:

Relocatable file:	If no errors were detected in the source file (called FILENAME:source), a relocatable file (called FILENAME:reloc) will be created. This file will be used by the linker to create an executable absolute file.
Listing file:	If a listfile is specified when the compiler is evoked, a file containing source lines with line numbers, program counter, level numbers, errors and expanded code (if specified) will be generated.
Assembly file:	If assembly file is specified by the option ASM_FILE anywhere in the source file, an assembly file, ASM6809, will be created in the current userid. This file will contain the assembly language source equivalent of the C program being compiled with the C language source intermixed as comments. This file may be assembled by the assembler.
Assembler symbol file:	This file is created unless the option $ASMB_SYM$ is disabled within the source program. An assembler symbol file (called <FILE_name>:asmb_sym) contains the symbolic information useful for program debugging during emulation.

Linking

After all program modules have been compiled (or assembled), the modules may be linked to form an executable absolute file. The compiler generates calls to a set of library routines for commonly used operations such as multiply, divide, comparisons, array referencing, etc. These routines must be linked with the program modules. There are three standard libraries which may be used to supply these operations. In addition, if real number arithmetic is used, one of the two real number libraries must also be linked.

The first is a debug library file called DLIB_6809:C6809. This library of relocatable procedures contains some extra code to detect errors such as division by 0, or overflow on multiplication.

The second library is called LIB_6809:C6809. This library, which has only a limited set of error-detection code, should execute slightly faster and take up less space in memory. This library may be linked in place of the debug library after reasonable assurance that the code is error free.

Linking with Real Numbers

When using real numbers for the 6809, the user must link with the real number support library. RealLIB:C6809.

The library, RealLIB:C6809, supports the C/64000 implementation of the IEEE real number standard for both long and short floating point numbers (C data types "double" and "float"). To allow for mixed "float" and "double" expressions, all internal real operations are performed using an unpacked real number format using a 64-bit mantissa (fraction), a separate sign bit, and a 16-bit signed exponent.

Both real number libraries will load subroutines in the PROG relocatable area. These libraries use the DATA relocatable area only for two reasons: a default stack area and a message buffer for error detection.

If the user does not supply his/her own versions of the real error reporting routines, INVALID and OVERFLOW, the real library will supply them and a DATA relocatable buffer area for reporting the error condition. See the section on real number libraries in Chapter 4 for more information on real number error detection.

The linker is evoked and the questions asked should be answered as follows:

link ...

Object files: MODULE1,MODULE2

Library files: DLIB_6809:C6809

.
.
.

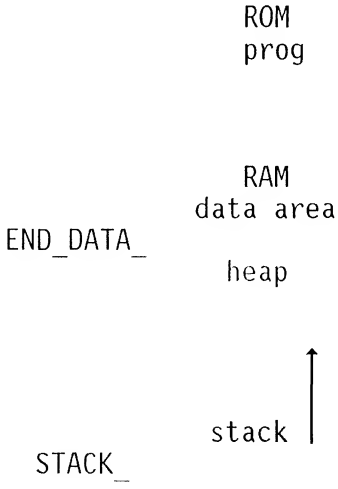
Absolute file name: PROGRAM

In the link listfile, the library routines that are referenced by the compiled code are linked at the end of the last user relocatable PROG and/or DATA areas. This fact must be considered for the proper choice of the stack pointer location, and PROG and DATA link addresses.

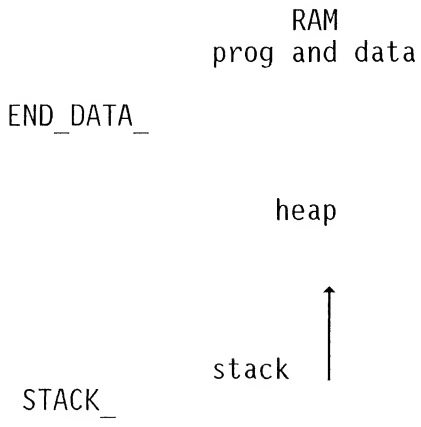
Emulation of C Programs

After all modules have been compiled (or assembled) and linked, the absolute file may be executed using the emulation facilities of the Model 64000. The emulator is initialized with the memory mapped in keeping with the target system and the stack pointer initialization in the code.

A program which is designed to run in read-only memory (ROM) should have been compiled with the $SEPARATE$ option. The memory should be mapped to have ROM and RAM as illustrated below.



For a program that is designed to run completely in random access memory (RAM), the memory mapping should look like the following:



If a transfer address was defined in the linking process, the emulation subsystem will remember the transfer address from the absolute file. The emulator will start execution of the program with the simple command:

```
run
```

or

```
step
```

If the program absolute file does not contain a transfer address these commands will start execution at address 0000H. The user must initialize the stack pointer register to his stack area before any program using subroutine calls is executed.

For C programs, stack initialization is done by the library routine "entry" or "ENTRY". (See Chapter 2 for additional run-time stack information). If the options $ENTRY ON, UPPER_KEYS OFF$ are used in the module defining the C function "main", the library will supply the routine "entry" which will initialize the stack and call the user routine "main". In this case, program execution in emulation should be performed using the command:

```
run from Entry
```

If the options $ENTRY ON, UPPER_KEYS ON$ are used in the module defining the C function "MAIN", the library will supply the routine "ENTRY" which will initialize the stack and call the user routine "MAIN". In this case, program execution in emulation should be performed using the command:

```
run from ENTRY
```

If the program runs to completion, the 6809 will remain in a loop at Z_END_PROGRAM:

```
Z_END_PROGRAM      BRA Z_END_PROGRAM
```

NOTE

It is important to remember that during emulation of C/64000 programs, a C compiler program may be debugged symbolically (using global symbols in the source program) or by source program line numbers of the form: #1. This is a feature that provides a powerful tool for emulation.

Debugging with DLIB_6809:C6809 Library

When initializing the emulator, it is a good idea to answer yes to the "stop processor on illegal opcode?" question since execution errors may result in a jump into the error handler file, Derrors:C6809.

If, while watching the execution of the code, the status line should indicate "illegal opcode executed at address XXXXH", note the address and enter the command:

```
display local_symbols_in Derrors:C6809
```

The list will roll off the screen; do not stop it with the reset key, since the information which rolls off is not important. When the list has stopped, scan the upper portion of the list for the address at which the illegal opcode occurred. The error type will be listed at the left of this address. (Descriptions of run time errors are given in Appendix A.) The list will also be generated when using library LIB_6809:C6809 by entering the following command:

```
display local_symbols_in Zerrors:C6809
```

The display will now appear as follows:

NOTE

The addresses will change depending upon the link.

Label	Address	Data	
Z_END_PROGRAM	1242H	C3H	
Z_ERR_RANGE	1270H	22H	Scan this portion for
Z_ERR_CASE	1258H	08H	where the illegal
Z_ERR_DIV_BY_0	124AH	08H	opcode occurred. The
Z_ERR_HEAP	1268H	08H	data field in this
Z_ERR_OVERFLOW	123CH	08H	portion is not
Z_ERR_SET_CONV	1251H	08H	significant.
Z_ERR_UNDERFLOW	1243H	08H	
Z_ERR_STRING	1235H	00H	
Z_CC_FLAGS	1296H	89H	
Z_ACC_A	1297H	8FH	The data field in
Z_ACC_B	1299H	F6H	portion may contain
Z_REG_X	1298H	F5H	information. The
Z_REG_U	129BH	F0H	addresses in this
Z_CALLER_H	1295H	69H	portion are not
Z_CALLER_L	1294H	A0H	significant.

Some of the errors will load locations with register and stack information.

NOTE

This compiler can generate duplicate symbols in the assembler symbol file for legal C programs. These symbols can be generated by function names that conflict with labels generated by the compiler, i.e. E, R, C, and D function labels. Refer to the C Compiler Reference Manual for a description of these labels.

These duplicate symbols can cause ambiguities with some HP 64000 logic analyzer measurements since a reference to a duplicated label may produce an incorrect result.

The compiler produces a warning message whenever it generates a duplicate label to warn the user that the use of that symbol in an analysis product may result in an incorrect address being traced. This potential problem can be solved by changing one of the duplicate function names, or by moving one of the functions to another file.

Example Warnings:

```
*****WARNING ?? - Symbol: Y, is duplicated in the asmb_sym file.
*****WARNING  ?? - Symbol: RY, is duplicated in the asmb_sym file.
```

NOTES

Chapter 2
C/64000 PROGRAMMING
6809

PROGRAMMING CONSIDERATIONS

Introduction

This chapter covers some important requirements of the run-time environment for 6809 C/64000 programs. Although some requirements may not be necessary for every program, the programmer should become familiar with the information supplied in order to use it when the structure of a 6809 program requires it.

Direct Addressing Mode

The 6809 direct page register (DP) is concatenated with any 6809 direct access address to generate the complete run-time address of an object. For example, if the instruction

```
LDA  <25H
```

is generated as a direct addressing instruction, the object that will be loaded into register A will be found at address 25H if the direct page register contents equal 00H. If the direct page register contents equal 0FEH, for example, then the object that will be loaded into register A will be found at address 0FE25H.

The DP register will be initialized to 00H by a 6809 Restart Interrupt. It will never be modified by the 6809 compiler.

The 6809 compiler will generate direct addressing instructions for any object known by the compiler to be located within the address range 00H and 0FFH. For the following C variable declaration:

```
$ORG = 20H$
 int FLAG;
 int INFORMATION;
$END_ORG$
```

the 6809 compiler will generate direct addressing instructions to access variables FLAG and INFORMATION, since their addresses are known by the compiler to be between 00H and 0FFH.

Program Initialization and Exit

The routine "entry", which gets loaded if the $ENTRY$ directive is ON and the function "main" is declared, appears as follows:

```
"6809"
    EXTERNAL MAIN
ENTRY
    LDS   #STACK_
    LDX   #0
    PSHS  X
    LBSR  MAIN
    LBSR  Z_END_PROGRAM
 EENTRY          EQU $-1
    GLOBAL   EENTRY
    GLOBAL  ENTRY
    EXTERNAL Z_END_PROGRAM
    EXTERNAL STACK
    END     ENTRY
```

```
"6809"
    EXTERNAL main
Entry
    LDS   #STACK-
    LDX   #0
    PSHS  X
    LBSR  main
    LBSR  Z_END_PROGRAM
entry   EQU  Entry00_D+00000H
Entry00_D
     RMB   00002H
EEntry          EQU $-1
    GLOBAL    EEntry
    GLOBAL    Entry
    GLOBAL    entry
    EXTERNAL  E_END_PROGRAM
    EXTERNAL  STACK_
 END       Entry
```

NOTE

If the $UPPER_KEYS$ directive was ON, the routine ENTRY will call MAIN.

Stack Pointer Initialization

The stack pointer is a hardware register maintained by the processor. Prior to use, however, it must be initialized by the user. A program that has a main code section must generate the following stack initialization statements in the relocatable file:

```
    EXT STACK_
LDS #STACK_
```

Since the EXT statement implies that the label STACK_ has been declared global (GLB) by another program module, the compiler will build a relocatable file, leaving assignment of the STACK_ value for the linker.

If the label STACK_ has not been declared global by any program module, the linker will search the applicable library for a default value. Depending upon which library has been selected by the user, one of the following default values will be selected:

a. If the DLIB_6809:C6809 library is linked, the stack will be assigned 512 bytes in the program (PROG) area of the linked modules.

b. If the LIB_6809:C6809 library is linked, the stack will be assigned 512 bytes in the data (DATA) area of the linked modules.

NOTE

Whenever the LIB_6809:C6809 library is linked, a DATA area location must be specified.

The user should allocate a larger stack when necessary. In particular, recursive programming will generally require a much larger stack than static programming to run properly.

Another approach to stack pointer initialization is to define a global variable called STACK_ as shown in the following example:

```
(file MODULE1:source)
        .
        .
        .
        $ORG   3F80H$
        Short STACK_AREA [127];
        Short STACK_;
        $END_ORG$
```

The compiler will generate relocatable code which sets the stack pointer to the address of STACK- (4000H in this example), and use an area of 128 bytes (3F80H..4000H) for the stack.

The use of an absolute address for the stack as in the above example has the user convenience of assigning a fixed block of memory for the stack. It may be better, however, to allow the compiler to actually preserve a relocatable data area for the stack by leaving out the ORG and END_ORG options. This will help prevent accidental reuse of the assigned stack area by another module.

An approach when linking assembly language files is to include the initial stack pointer value or a stack area in an assembly file such as:

```
"6809"
            GLB STACK_ .
STACK_      EQU 2000H    ;puts initial stack
            .            ; pointer at 2000H
            .
            .
```

or:

```
"6809"
            GLB STACK_ .
            DATA
 STACKBOT   RMB <stacksize>  ;puts stack
                             ; storage in the
 STACK_:    RMB 1            ; DATA area of
            .                ; the program
            .
            .
```

Note that the address of STACK_ will receive the first data byte being pushed. This file may then be linked with the other program modules generated by the compiler as follows:

```
Object files:   ASMFILE1,MODULE1,MODULE2....
```

After stack initialization the routine "main" is called. On return, a jump to Z_END_PROGRAM is done.

If the $ENTRY$ directive was turned OFF, the user is responsible for initialization.

Stack Format During Program Execution

Integer values are pushed on the stack: low_byte,high_byte, and popped: high_byte,low_byte.

```
Execution of PSHS X:

       Stack before execution:


                    :
                    .              <         (S)


       Stack after execution:


                    :
                    .

              Low_byte  of (X)

              High_byte of (X)     <         (S)


Execution of PULS X:

       Stack before execution:


                    :
                    .

              Low_byte  of (X)

              High_byte of (X)     <         (S)


      Stack after execution:


                    :
                    .              <         (S)
```

When any recursive routine is entered in the 6809 compiler, the library routine RENTRY_ or VRENTRY_ (variable number of parameters) is called. RENTRY_ or VRENTRY_ will allocate the routine's data area and will copy and arrange the parameters. In addition, it will create the necessary static links.

The following sample program illustrates some principles of stack organization:

```
 1  "C"
 2  "6809"
 3  $RECURSIVE OFF$       /*NOTE: RECURSIVE IS ON BY DEFAULT*/
 4
 5  static_func(p1)
 6  {
 7  int p1;
 8
 9      p1  =  1;
10  }
11
12  $RECURSIVE ON$
13
14  recursive_func(p1)
15  {
16  int p1;
17
18      p1 = 1;
19  }
20
21  little_func(p1)
22  {
23      return p1;
24  }
25
26  struct big_type {
27     int i1,i2,i3;
28  }bt1,bt2;
29
30  struct big_type big_func(p1)
31  struct big_type p1;
32  {
33     return p1;
34  }
35
36  int i,j;
37
38  main ( )
39  {
40     static_func(i);
41     recursive_func(i);
42     i = little_func(j);
43     bt1 = big_func(bt2);
44  }
```

When executing line 9, the stack appears as follows:

Return address to main <---Stack Pointer

NOTE

There is no data area on the stack because static_func is a static procedure.

When executing line 18, the stack appears as follows:

return address to main

recursive_func's Parameters

No. of parameter bytes

recursive_func's Data <---Stack Pointer

When executing line 23, the stack appears as follows:

Return address to main

big-func's Parameters

No. of parameter bytes <---Stack Pointer

When executing line 33, the stack appears as follows:

Return address to main

big_func's Parameters

Address of bt1

No. of parameter bytes <---Stack Pointer

Note that for big_func, the address of the result (bt1) is passed since it is greater than two bytes. For little_func, the address of the result (i) is not passed as the size is only two bytes and the result is returned to a register.

Recursive Routines - Calling and Returning Sequences

Fixed Number of Parameters. The calling sequence for a recursive routine with a fixed number of parameters is as follows:

If the parameters are passed in the registers (see PARAM_), the calling sequence will be:

```
LDr1 par#1
LDr2 par#2
 :
 .
LDrn par#n
LBSR receiving_routine
```

and the receiving routine will push the parameters when it is entered, and then call RENTRY_.

If the parameters are not passed in registers, the calling sequence is as follows:

```
LDr1 par#1
LDr2 par#2
 :
 .
LDrX par#x
PSHS r1,r2,r3,...rX,PC
LDr1 par#x+1
LDr2 par#x+2
 :
 .
LDrX par#n
PSHS r1,r2,r3,...rX
LBSR receiving_routine
```

and the receiving routine will only call RENTRY_.

When returning from RENTRY_, the stack format at exit will be as follows:

RA(calling routine)

Par.'s

Var.'s

Static Links

< (S)

The returning sequence is as follows:

```
LEAS  var_area_size+par_area_size+level*2,S
RTS
```

Variable Number of Parameters. The parameters are never passed in registers for a variable number of parameters call. The calling sequence is:

```
LDr1     par#1
LDr2     par#2
  .
  .
  .
  .
LDrx     par#x
PSHS     r1,r2,...,rx
LDr1     par#x+1
LDr2     par#x+2
  .
  .
  .
LDrx-1   par#n
LDrx     #no_of_par_bytes
PSHS     r1,r2,...,rx
LBSR     receiving routine
LEAS     no_of_par_bytes+2,S
```

The receiving routine will only call VRENTRY_. When returning from VRENTRY_, the stack appears as follows:

Parameters

No. of parameter bytes

Variables

<----(S)

Interrupt Vector Handling

The run-time programming environment of C/64000 programs on the 6809 processor has been designed to impose a minimum amount of constraints on the user. As a result the code produced by the compiler is safely interruptable as long as the interrupt driven process restores the registers (which have been automatically pushed onto the stack when the 6809 recognized the interrupt) with a return from interrupt (RTI) instruction.

The 6809 processor supports four types of interrupts: a reset (or powerup) interrupt, a non-maskable interrupt, a maskable interrupt, and a software interrupt. The first three of these are enabled by external control signals to the processor, while the last one is enabled by software program control. When the processor detects one of these interrupts it saves the current status of the processor and jumps to the address in the interrupt vector for that type of interrupt. These vectors are in the last 13 bytes of memory.

For the rest of this discussion assume that the following assembly module defines the interrupt vectors.

```
FILE: IRQ:C6809            HEWLETT-PACKARD: 6809 Assembler

LOCATION OBJECT CODE LINE     SOURCE LINE

                      1 "6809"
                      2  NAME "Interrupt Vector Definition"
                      3
                      4  EXT SOFT_INT_3, SOFT_INT_2, SOFT_INT_1
                      5  EXT FIRQ_INT, IRQ_INT, NMI_INT, RESTART_INT
                      6
                      7  ORG OFFF2H
                      8
 FFF2    0000         9  FDB SOFT_INT_3
                     10
 FFF4    0000        11  FDB SOFT_INT_2
                     12
 FFF6    0000        13  FDB FIRQ_INT
                     14
 FFF8    0000        15  FDB IRQ_INT
                     16
 FFFA    0000        17  FDB SOFT_INT_1
                     18
 FFFC    0000        19  FDB NMI_INT
                     20
 FFFE    0000        21  FDB RESTART_INT

Errors=    0
```

The C/64000 ENTRY program may logically be used as the RESTART_INT to be called on RESTART interrupt. A main program initializes the run time environment for C program execution and ends with the jump to a tight loop at Z_END_PROGRAM (generated by the compiler), thus fitting all the requirements of the RESTART_INT routine.

C/64000 allows the user to define procedures as routines to be called in the interrupt vector by using the $INTERRUPT ON$ option. The $INTERRUPT$ option is only recognized for procedures defined at the outer block of a program. An interrupt procedure needs to be declared global so its address can be available at link time to load the proper interrupt vector. Nothing special is done upon entry to the $INTERRUPT$ procedure. At the end of the procedure the compiler generates a return from interrupt (RTI) instruction instead of a return from subroutine instruction (RTS). An $INTERRUPT$ procedure may not be called like a normal C/64000 procedure because of the RTI return instruction.

The interrupt procedure can have no parameters but it may be compiled in either the $RECURSIVE ON$ or $RECURSIVE OFF$ modes. The $RECURSIVE ON$ mode is required if it is possible to be processing multiple interrupts at the same time.

Any special treatment of interrupts would require some assembly language modules since instructions associated with interrupts are not available in C (SYNC, CWAI, ORCC, ANDCC).

With the previously defined interrupt vector definition the user should compile procedures IRQ_INT, NMI_INT, and SOFT_INT with the $INTERRUPT ON$ option enabled. Care must be taken to turn off this option explicitly.

USER DEFINED OPERATORS

General

C/64000 allows the user to define his own special operators (user defined operators). User defined operators are created by using the option: $USER_DEFINED$ during the declaration of a user type. The option will apply to the declaration of one user type.

For user defined operators, the compiler will not generate in-line code to perform the operations, instead, it will generate calls to user provided run-time routines. The run-time routine names will be a composite of the user's type name and the operation being performed: TYPENAME_OPERATION. The first eleven characters of the user's type name are concatenated with an underscore and three characters identifying the operation.

Operations

The following is a list of operators that can be user defined and the run-time routine names that the compiler will create when the operations are used on a user type:

	Operation	Symbol	Run-time Routine
1.	Add	+	<typename>_ADD
2.	Negate	-	<typename>_NEG
3.	Subtract	-	<typename>_SUB
4.	Multiply	*	<typename>_MUL
5.	Divide	/	<typename>_DIV
6.	Modulus	%	<typename>_MOD
7.	Equal Comparison	==	<typename>_EQU
8.	Not Equal Comparison	!=	<typename>_NEQ
9.	Less Than or Equal to Comparison	<=	<typename>_LEQ
10.	Greater Than or Equal to Comparison	>=	<typename>_GEQ
11.	Less Than Comparison	<	<typename>_LES
12.	Greater Than Comparison	>	<typename>_GTR

The compiler will provide the user with a Store routine. The 6809 compiler will use the multi-byte move routine (MBmove).

Parameters

The parameters are passed to a routine by reference, i.e., the addresses of the parameters are passed. For the 6809, the parameters are passed in the following registers:

Input: D contains the address of the first parameter
X contains the address of the second parameter
Y contains the address of the third parameter

Output: The result should be assigned through register Y

Register Y will not be defined for relational operations. The result should be assigned to register B, and the Z flag in register CC should be set according to the following:

TRUE - B set to 1, Z flag is set

FALSE - B set to 0, Z flag is reset

Register U will not be defined for the unary operation Negate; register X will contain the result.

The routines for 6809 user-defined operators can be written in C; routines can be either static or recursive.

To use C procedures with user_defined operators, the option $FIXED_PARAMETERS$ should be used.

Example:

```
$FIXED_PARAMETERS ON$

USER_MUL (OPERAND1, OPERAND2, RESULT)

USER *OPERAND1, *OPERAND2, *RESULT;

$FIXED PARAMETERS OFF$
```

The routines for binary operators can be defined in two ways: (1) a procedure with 3 VAR parameters: argument1, argument2, and result, or (2) as a function with 2 parameters (defined as the first two parameters in case 1) and the result being the function result. Argument1 always corresponds to the left-hand side operand and argument2 to the right-hand side operand as follows:

result = argument1 OPERATOR argument2

The unary operator Negate routine can be defined as a procedure with two VAR parameters, or as a function with one parameter. The first parameter is the argument and the result is either the second parameter or the function result.

Finally, the routines for relational operators should be defined as functions with two VAR parameters: argument1 and argument2 and a result of type boolean. Care should be taken to assure that the Z flag is set to the correct value. This may be accomplished by making certain that the last thing the function does is to assign the function result.

Example:

The following program defines and uses the user type "MATRIX":

```
 1  0000  0   "C"
                  EXTERNAL mat1
                  EXTERNAL mat2
                  EXTERNAL mat3
                  EXTERNAL flag
                  EXTERNAL entry
 2  0000  0   "6809"
 3  0000  0   $RECURSIVE OFF$
 4  0000  0   #define MAXSIZE 5
 5  0000  0   $USER_DEFINED$
 6  0000  0   struct {
 7  0000  1      int mat[MAXSIZE][MAXSIZE]; /*maximum size*/
 +  0000  1      int mat[ 5][ 5];
 8  0000  1      short nrows,ncolumns;      /*actual size*/
 9  0000  1   }  typedef MATRIX;      /*user defined MATRIX*/
10  0000  0
11  0000  0   extern MATRIX mat1,mat2,mat3; /*extern causes names
12  0000  0                               to appear in listing*/
13  0000  0   extern int flag;
14  0000  0
15  0000  0   main()
16  0000  0   {
       0000          main
```

```
17  0000  1    mat1 = mat2 + mat3 * mat1;
      0000              LDX   #mat1
      0003              LDY   #main01_D
      0007              LDD   #mat3
      000A              LBSR  MATRIX_MUL
      000D              LDX   #main01_D
      0010              LDY   #mat1
      0014              LDD   #mat2
      0017              LBSR  MATRIX_ADD
18  001A  1    if (mat1 == mat2) mat1 = mat2;
      001A              LDX   #mat2
      001D              LDD   #mat1
      0020              LBSR  MATRIX_EQU
      0023              LBEQ  main01_L1
      0027              LDX   #mat2
      002A              LDU   #mat1
      002D              LDD   #00034H
      0030              LBSR  MBmove
19  0033  1    flag = mat1 != mat2;
      0033          main01_L1
      0033              LDX   #mat2
      0036              LDD   #mat1
      0039              LBSR  MATRIX_NEQ
      003C              CLRA
      003D              STD   flag
20  0040  1
      0040              RTS
      0041          main01_D
      0041              RMB   00034H
      0041          Emain             EQU $-1
      0041
      0041              GLOBAL     Emain
      0041

                        EXTERNAL  MATRIX_ADD
                        EXTERNAL  MATRIX_MUL
                        EXTERNAL  MATRIX_EQU
                        EXTERNAL  MATRIX_NEQ
                        GLOBAL    main
                        EXTERNAL  MBmove
```

OPTIONS

OPTIMIZE
Default OFF.

Forward Branches - The 6809 has short branch instructions that can be used when the location to be branched to is within 128 bytes from the branch location. The compiler optimizes all backward branches since it knows the distance to be branched to at compile time. Forward branches, on the other hand, are always assumed to be long (the distance to be branched to is not known). Since most forward branches have been found to be short, an optimization has been added to the 6809 compiler such that when the option $OPTIMIZE$ is ON, the compiler assumes that all forward branches are short. This will cause a compiler error (Pass 3 ERROR-- 1200) if the branch happens to be out of range; if this occurs, $OPTIMIZE$ should be turned OFF around the branch where the error occurs. The error reads: Long range error, turn off OPTIMIZE for this line.

Recursive Parameter Addressing - The 6809 has three different stack offset sizes: 5 bits, 8 bits, and 16 bits. The parameters on a recursive routine are always allocated on the stack and their stack offsets will depend on the number and size of the variables and temporaries used in the routine. Since the stack offset size for the parameters is not known until the end of the routine, all stack offsets for recursive parameters are assumed to be 16 bits long. When the option $OPTIMIZE$ is ON, however, the compiler makes an "educated guess" on the size of the parameter stack offset. This can cause a compiler error if the actual offset size does not match the compilers guess, and the compiler program counter will differ from the actual program counter. The error will appear on the next label following the statement where the parameter is accessed. When this error occurs, turn OPTIMIZE OFF at least for the statement where the parameter is accessed. The error has a rippling effect, so it is recommended that $OPTIMIZE$ be turned OFF for the entire routine. The error reads: Program counters do not agree.

DEBUG
Default OFF.

The DEBUG option is used to check for arithmetic errors on arithmetic operations for the standard types. Operations which may normally be performed with in-line code (such as a BYTE add), will be performed using a subroutine call if the $DEBUG$ option is ON. The library routines in the debug library have checks to detect arithmetic errors. The routines of the same name in the non-debug libraries perform the same arithmetic operation but do not detect any error conditions.

FIXED_PARAMETERS
Default OFF.

This option allows the user to define a routine with a fixed number of parameters. This definition is necessary when interacting with Pascal programs. With variable number of parameters, the parameters are never passed in registers and an extra parameter, indicating the number of parameter bytes, is required. There is an extra overhead for clearing the parameters from the stack at the calling end instead of at the receiving end.

POSITION INDEPENDENT CODE

Parts of the 6809 compiler are not position independent.

Switch-statements have position-dependent code (absolute jump). This can be avoided by writing if-statements instead of switch-statements.

Addressing of Constants

Some addresses and dope vectors are allocated in a constant data area with the label "CONST_prog". Possible cases that require constants are: the use of the "&" function, multi-dimensional or integer element array references, pointers, user-defined constants, etc. Compile your program with options expand and check the assembly code for the label "CONST_prog" to verify that the compiler has used the constant area. The constant area is always allocated at the end of the main program.

Addressing of Static Variables and Temporaries

A program with no static variables and all procedures being recursive will not have this problem.

External Procedures

Any external procedure requires an address (determined by the linker); this includes the run-time routines.

PASS 2 AND PASS 3 ERRORS

Pass 2 and 3 errors will be displayed on the screen with the message:

LINE # <line number>--PASS2 ERROR # <error number>
or LINE # <line number>--PASS3 ERROR # <error number>

In addition, if a listing file has been indicated for the compilation it will indicate pass 2 and 3 errors where they occurred. It will also give you a listing of the meaning of each error.

Pass 2 and 3 error numbers will always be >=1000. Errors with numbers between 1000 and 1099 are fatal errors. Errors with numbers >=1100 are non-fatal errors.

Pass 2 and 3 will stop generating code after a fatal pass 2 or 3 error. If a listing file has been indicated for the compilation, pass 3 will give you a listing with errors. Non-fatal errors are output to the display and to the listing file (if one exists), but compilation continues after appropriate action has been taken to correct the error. A list of pass 2 and 3 errors is given in Table 2-1.

Table 2-1. 6809 Pass 2 And Pass 3 Errors

1000 - "Out of memory"
The 6809 code generator has run out of memory, break up your program and recompile.

1001 - "Size not implemented"
An integer larger than 16 bits has been detected.

1002 - "Size error"
A size larger than the maximum size allowed for a type has been detected.

1003 - "Type not implemented"
A real or other unimplemented type has been detected.

1004 - "Type error"
An operation with an incorrect type of operands has been detected; for example, a negation of an unsigned value.

1005 - "Unimplemented feature"
An attempt has been made at using a feature not implemented on the 6809 code generator.

1006 - "Compiler error. Contact Hewlett-Packard"
This is a compiler level error. Please report this error to Hewlett-Packard as soon as possible.

Table 2-1. 6809 Pass 2 And Pass 3 Errors (Cont'd)

1007 - "Expression too complicated"
The compiler can not handle the level of complexity of this expression, simplify your expression.

1008 - "Register needed but not available"
The compiler can not generate more code without additional registers; add temporary results for your operations.

1010 - "Too many constants"
More than 256 constant values required during code generation. Eliminate duplicate real constants or break up module and recompile.

1103 - "Interrupt procedure must not have parameters"
An interrupt procedure can not have parameters. The compiler will ignore the parameters and continue to generate code.

1104 - "Interrupt procedure call not allowed"
An interrupt routine can only be accessed through an interrupt vector, since it will return with an RTI instead of an RTS. The compiler will ignore calls to interrupt routines.

1106 - "Program counter overflow"
The program will wrap around 0FFFFH. Other errors may occur if this is not corrected.

1107 - "Data counter overflow"
The data counter will wrap around 0FFFFH.

1110 - "Defined a static routine within a recursive one"
Static routines can not be defined within recursive routines because of the difference in addressing. The compiler makes the routine recursive and continues to generate code.

1111 - "Interrupt routines must be at level one"
All interrupt routines must be at level one. For routines defined at levels greater than 1 with $INTERRUPT ON$, the compiler will ignore the option, i.e. it will generate a non-interrupt routine.

1113 - "Program counters do not agree"
The program counter for a label generated by Pass 2 does not agree with the program counter for that label in Pass 3. Please report the error to Hewlett-Packard as soon as possible. This error is detected in Pass 3.

1200 - "Long range error; turn off OPTIMIZE for this line"
The option $OPTIMIZE$ causes the code generator to use 2-byte branch instructions for forward branches. This error occurs when the label is too far away. Turning $OPTIMIZE OFF$ for this line of code will produce a long jump which will always work.

NOTES

Chapter 3

RUN-TIME LIBRARY SPECIFICATIONS

General

This chapter describes the run-time library routines available in the C/64000 compiler library for the 6809 microprocessor. Each routine description includes the purpose, input requirements, and output results.

The library is logically divided into two groups of routines. One group contains the standard library procedures and functions. The second group supplies the elementary routines that supplement the standard 6809 instruction set. Tables 3-1 and 3-2 list the standard and supplemental routines for the 6809 microprocessor.

NOTE

Some of the libraries listed in the tables will never be accessed by the "C" compiler but are available in the run-time library for the user.

Table 3-1. Library Routines (Standard)

Name	Purpose	Ref Page
ARRAY_	Compute address of array element	3-6
PARAM_	Pass parameters to procedures	3-14
VPARAM_	Pass variable parameters to procedures	3-16
RENTRY_	Recursive procedure entry	3-11
VRENTRY_	Recursive procedure entry	3-13
INITHEAP_	Declares block of memory as memory pool	3-10
NEW	Dynamic memory allocation	3-10
DISPOSE	Dynamic memory deallocation	3-10
MARK	Save current status of dynamic memory heap	3-10
RELEASE	Restore prior status of dynamic memory heap	3-10

Table 3-2. Library Routines (for 6809)

8-bit Arithmetic Group

Name	Purpose	Ref Page
Zbyteabs	Byte absolute value	3-18
Zbyteneg	Byte negation	3-18
Zbyteadd	Byte addition	3-18
Zubyteadd	Unsigned byte addition	3-19
Zbytesub	Byte subtraction	3-19
Zubytesub	Unsigned byte subtraction	3-19
Zbytemul	Byte multiplication	3-19
Zbytediv	Byte division	3-18
Zubytediv	Unsigned byte division	3-19

16-Bit Arithmetic Group

Name	Purpose	Ref Page
Zintadd	Integer addition	3-20
Zuintadd	Unsigned integer addition	3-21
Zintsub	Integer subtraction	3-21
Zuintsub	Unsigned integer subtraction	3-22
Zintmul	Integer multiplication	3-21
Zuintmul	Unsigned integer multiplication	3-22
Zintdiv	Integer division	3-21
Zuintdiv	Unsigned integer division	3-21

Byte and Word Shifts

Name	Purpose	Ref Page
Zbshift	Byte shift logical with zero fill	3-23
Zbrotate	Byte shift circular	3-23
Zwshift	Word shift logical with zero fill	3-23
Zwrotate	Word shift circular	3-23

Table 3-2. Library Routines (for 6809)(Cont'd)

Byte and Word Set Operations

Name	Purpose	Ref Page
Zbinset8	Byte in 8-bit set	3-24
Zbinset16	Byte in 16-bit set	3-25
Zbtoset8	Byte to 8-bit set	3-24
Zbtoset16	Byte to 16-bit set	3-25
Zwinset16	Word in 16-bit set	3-26
Zwtoset16	Word to 16-bit set	3-26

Multi-byte Operations

Name	Purpose	Ref Page
MBmove	Multi-byte assignment	3-27
MBequ	Multi-byte equality test	3-27
MBneq	Multi-byte inequality test	3-27
MBgeq	Multi-byte greater than or equal test	3-27
MBgtr	Multi-byte greater than test	3-27
MBleq	Multi-byte less than or equal test	3-27
MBles	Multi-byte less than test	3-27

Multi-byte Set Operations

Name	Purpose	Ref Page
INSETmb	Multi-byte set inclusion	3-29
TOSETmb	Multi-byte set formation	3-29
SETmbINT	Multi-byte set intersection	3-29
SETmbUNI	Multi-byte set union	3-29
SETmbDIF	Multi-byte set difference or equal	3-30
SETmbLEQ	Multi-byte set less than or equal	3-30

Table 3-2. Library Routines (6809)(Cont'd)

Byte and Integer Comparison and Bounds Checking Routines

Name	Purpose	Ref Page
Zcc	Carry cleared test	3-32
Zequ	Byte and integer equality test	3-32
Zneq	Byte and integer inequality test	3-32
Zgeq	Byte and integer greater than or equal test	3-32
Zgtr	Byte and integer greater than test	3-32
Zleq	Byte and integer less than or equal test	3-32
Zles	Byte and integer less than test	3-32
Zugeq	Unsigned byte and integer greater than or equal test	3-32
Zugtr	Unsigned byte and integer greater than test	3-32
Zuleq	Unsigned byte and integer less than or equal test	3-32
Zules	Unsigned byte and integer less than test	3-32
Zbbounds	Byte bounds checking	3-33
Zubbounds	Unsigned byte bounds checking	3-33
Zwbounds	Integer bounds checking	3-33
Zuwbounds	Unsigned integer bounds checking	3-33

Table 3-2. Library Routines (for 6809)(Cont'd)

String Operations

Name	Purpose	Ref Page
STmove	String assignment	3-36
STequ	String equality test	3-34
STneq	String inequality test	3-34
STgeq	String greater than or equal test	3-34
STgtr	String greater than test	3-34
STleq	String less than or equal test	3-34
STles	String less than test	3-34
CHequ	String-char equality test	3-34
CHneq	String-char inequality test	3-34
CHgeq	String-char greater than or equal test	3-34
CHgtr	String-char greater than test	3-34
CHleq	String-char less than or equal test	3-34
CHles	String-char less than test	3-34

Miscellaneous

Name	Purpose
END_DATA_	Label at the end of the library that can be used to allocate the HEAP area.
Z_END_PROGRAM	Label called at the end of the main program.
EMPTY_SET_	The largest possible empty set for the 6809.
STACK_	Label for stack
COMPB_	Compares bytes, called by the comparison routines.
TRUE_	Returns a true result (1) in register (B) and resets the Z flag. Called by the comparison routines.
FALSE_	Returns a false result (0) in register (B) and sets the Z flag. Called by the comparison routines.
MASTER	Memory allocation global

ARRAY REFERENCE ROUTINES

The C/64000 compiler supports generalized array references with up to 10 indices. The array reference routine is called with the parameters:

```
DOPE_VECTOR   -   address of a record describing the array.

BASE_ADDRESS  -   address of the first element of the
                  array.  (May be indirected like a VAR
                  parameter.)

Index_list    -   addresses of the actual index expressions
                  (one for each formal index expression).
```

The array reference routine return the computed memory address to the X register.

ARRAY_

The ARRAY_ routine returns the memory address of an n-dimensional array reference expression. The array reference call for the 3-index array variable expression:

A[I][J][7]

would be:

```
LDU  base_address          ; base address of array A
LDY  I
LDX  J
PSHS U,Y,X
LDU  #0007H
PSHS U
LDA  #3                    ; number of indices passed
LDX  dope_vector_address   ; for array A
LBSR ARRAY_
```

To illustrate the use of indirection required for the base address, consider variable B defined as a pointer to an array of the same type as A in the above example. A reference to an element of B with the variable array expression:

*B[6+Y][J][7]

would generate a call to ARRAY_ in the form:

```
LDD  Y
ADDD #0006H
LDU  [B]                        ; base address of array A
TFR  D,Y
LDX  J
PSHS U,Y,X
LDU  #0007H
PSHS U
LDA  #3                         ; number of indices passed
LDX  dope_vector_address        ; for array A
LBSR ARRAY_
```

Multi-dimensioned arrays are simply defined as array of arrays. An array may be referred to in its entirety (a so-called ENTIRE variable) by referring to the array by its name using no parameters. A variable expression allows the user to refer to an INDEXED element type by allowing from 1 to N index expressions to be used in an array reference. C arrays are stored such that the rightmost subscript changes faster.

The ARRAY_ call for a two-indexed array variable expression with a 3-dimensional array A is as follows:

A[I][J]

For example:

```
LDU  base_address          ; base address of array A
LDY  I
LDX  J
PSHS U,Y,X
LDA  #2                    ; number of indices passed
LDA  dope_vector_address   ; for array A
LBSR ARRAY_
```

The formulae for computing array element addresses are as follows:

a. The formula used to compute the array element address is:

ADDRESS: BASE_ADDRESS + (-OFFSET_CONSTANT) +
(I1 * PROD_1) + (I2 * PROD_2) +...+
(IN * PROD_N)

b. The (-OFFSET_CONSTANT) term is the product of the index lower bounds and the row widths, i.e.,

(I1L * PROD_1) + (I2L * PROD_2) +...+
(INL * PROD_N)

c. The expression used to compute the array row reference using N-1 index expression is:

```
row_address := BASE_ADDRESS + (-OFFSET_CONSTANT)
      + (I1 * PROD_1) +...+
      (InMINUS_1 * InMINUS_1) + ROWnMINUS_1
```

NOTE

The addition of ROWnMINUS_1 takes you to ROWn.

Generalized Array DOPE_VECTOR

The form of the general array reference dope vector is equivalent to:

```
DOPE_VECTOR       FDB N                    ;number of
                                           ; dimensions
                  FDB (-OFFSET_CONSTANT)   ;negative of
                                           ; constant
                  FDB PROD_1
                  FDB PROD_2
                  ...
                  FDB PROD_N
                  FDB ROW1
                  FDB ROW2
                  ...
                  FDB ROWnMINUS_1
```

About the Routine:

At termination, this routine returns the stack pointer to the location it held at the beginning of the program.

The array reference routines return the computed memory address in the X register.

NOTE

Users who write assembly language programs that define and use multi-dimension arrays to be used with the ARRAY_ routine need to ensure that their use is consistent with the C compiler. In order to accomplish this, it is recommended that the user write a simple C program defining and using the arrays. The user can then use the expanded listing file or the ASM_FILE option to determine how the C compiler accesses these arrays and defines the array dope vectors. It is important that the user's array dope vector be identical to that produced by the compiler.

DYNAMIC MEMORY ALLOCATION

C/64000 supports dynamic allocation and deallocation of storage space through the procedures NEW, DISPOSE, MARK, RELEASE, and INITHEAP. These routines must be declared external in the C program.

NOTE

These routines; NEW, DISPOSE, MARK, RELEASE, and INITHEAP, are static and should be defined with $RECURSIVE OFF$.

INITHEAP

The user declares a block of memory to be used as the memory pool or heap by calling: INITHEAP (Start_address, Length_in_bytes). The resultant heap will be six bytes smaller than length_in_bytes.

NEW

The procedure NEW (Pointer) is used to allocate space. The procedure, NEW, searches for available space in a free-list of ascending size blocks. When a block is found that is the proper size or larger, it is allocated and any space left over is returned to the free-list in a new place corresponding to the size of the leftover block. If the referenced block is four or less bytes in size, four bytes will be allocated.

DISPOSE

The procedure DISPOSE is exactly the reverse of the procedure NEW. It indicates that storage occuppied by the indicated variable is no longer required.

MARK

This procedure marks the state of the heap in the designated variable that may be of any pointer type. The variable must not be subsequently altered by assignment.

RELEASE

The procedure RELEASE restores the state of the heap to the value in the indicated variable. This will have the effect of disposing all heap objects created by the NEW procedure since the variable was marked. The variable must contain a value returned by a previous call to MARK; this value may not have been passed previously as a parameter to RELEASE.

RECURSIVE ENTRY

C/64000 supports recursive and reentrant calling sequences for procedures compiled for the 6809 by additional run-time entry code. This code causes the local data area of a procedure to be allocated onto the stack before entry to the procedure and to be deallocated from the stack upon exit from the procedure. These functions are performed by the procedures RENTRY_/VRENTRY_.

RENTRY_

RENTRY_ is called at the entry point of a procedure or function which has been declared with the option $RECURSIVE ON$. RENTRY_. will copy the parameters, set the static links, and allocate the variable size area.

RENTRY_ is called upon entry to a recursive C procedure or function. The calling sequence is:

```
LDU   var_area_size
LDA   #level                        ;always <=16
LDB   #register_parameters_flag     ; true (1) or
                                    ; false (0)
LDX   par_area_size
```

The stack format at entry to RENTRY_ is one of the following:

With no parameters:

RA (calling routine)

RA from RENTRY_
< (S)

With parameters passed in the registers:

RA (calling routine)

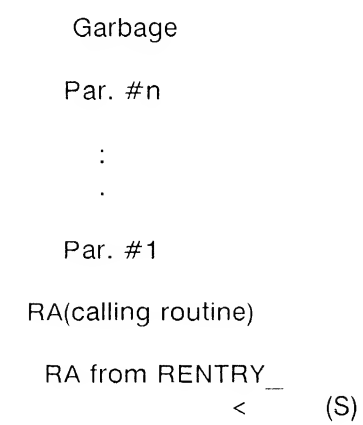
Par. #n

:
.

Par. #1

RA from RENTRY_
< (S)

With parameters passed on the stack:



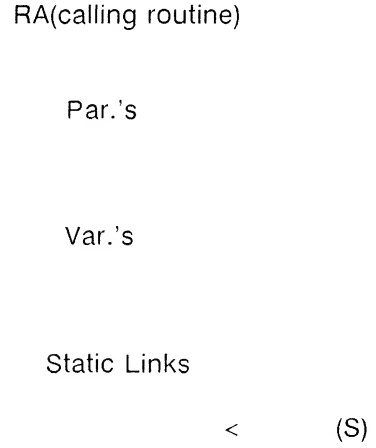
Procedure:

a. Copy the calling routine's RA to "Garbage" if necessary.

b. Allocate variables area.

c. Store the static links.

The stack format at exit from RENTRY_ is as follows:



All registers but CC are modified.

VRENTRY_

VRENTRY_ is called at the entry point of a recursive procedure or function with variable number of parameters (i.e. FIXED_PARAMETERS not ON). VRENTRY_ will allocate the variable size area.

VRENTRY_ is called upon entry to a recursive C procedure or function. The calling sequence is:

```
LDD   var_area_size
LBSR  VRENTRY_
```

The stack format at entry to RENTRY_ is one of the following:

With no parameters:

```
RA (calling routine)

RA from RENTRY_
                <        (S)
```

With parameters:

```
Par. #n

   :
   .

Par. #1

No. of parameter bytes

RA(calling routine)

RA from RENTRY_
                <        (S)
```

The stack format at exit from RENTRY_ is as follows:

```
Par.'s

RA (calling routine)

Var.'s
                <        (S)
```

All registers but CC are modified.

Parameter Passer (PARAM_)

The parameter passer is called from a static routine receiving parameters where the parameters are not passed in registers. PARAM_ transfers the parameters from the stack to the called routine's static data area.

If the total par-area-size in a static routine is less or equal to eight bytes and there are not more than two parameters of size equal to one byte, then the parameters will be passed in registers (instead of on the stack), else all the parameters will be pushed on the stack.

A receiving routine where parameters are passed in the registers has to store or push (for recursive routines) every parameter in the receiving routine's data area.

Calling sequence (from the routine passing the parameters, to the routine receiving the parameters) for parameters not passed in registers:

a. Calling a recursive routine:

```
LDr1   par#1
LDr2   par#2
 .
 .
 .
LDrX   par#x
PSHS   r1,r2,r3,...rX,PC
LDr1   par#x+1
LDr2   par#x+2
 .
 .
 .
LDrX      par#n
PSHS      r1,r2,r3,...,rX
LBSR      receiving_routine
```

b. Calling a static routine:

```
LDr1    par#1
LDr2    par#2
  .
  .
  .
LDrX    par#x
PSHS     r1,r2,r3,...,rX
LDr1    par#x+1
LDr2    par#x+2
  .
  .
  .
LDrX    par#n
PSHS     r1,r2,r3,...,rX
LBSR     receiving_routine
```

Calling sequence (from a static routine receiving the parameters, to PARAM_):

```
LDX   to_address
LDD   par_area_size
LBSR  PARAM_
```

Stack format at entry to PARAM_:

```
    Par. #n
      .
      .
      .
    Par. #1

  RA (calling routine)

 RA (receiving routine)
                    <      (S)
```

Stack format at exit from PARAM_:

```
  RA (calling routine)
                      <      (S)
```

NOTE

Users who write assembly language programs that define and use procedures and functions, particularly with parameters, need to ensure that their use is consistent with the C compiler. In order to accomplish this, it is recommended that the user write a simple C program defining the procedure or function with the desired parameter list and an empty program block for code. The user can then use the expanded listing file or the ASM_FILE option to determine how the C compiler enters and exists the equivalent do-nothing procedure and how the parameters are passed. It is important that the user's assembly language routines follow the same entry, parameter passing, and exit code produced by the compiler. In particular, it is important that recursive or static mode declarations (and use) be consistent.

VPARAM_

The variable parameter passer is the same as PARAM_ except that it includes an extra parameter that indicates the number of parameter bytes passed.

Calling a recursive routine:

```
LDr1   par#1
LDr2   par#2
  .
  .
  .
LDrX   par#x
PSHS   r1,r2,r3,...,rX
LDr1   par#x+1
LDr2   par#x+2
  .
  .
  .
LDrx-1   par#n
LDrx     #no_of_par_bytes
PSHS     r1,r2,r3,...,rx
LBSR     receiving_routine
```

The calling sequence (from a static routine ... VPARAM_):

```
LDX   to_address
LBSR  VPARAM_
```

The stack format at entry to VPARAM_ is as follows:

```
Par #n

  .
  .
  .

Par #1

No. of Parameter bytes

RA (calling)

RA (receiving)
                < (S)
```

The stack format at exit from VPARAM_ is as follows:

```
Par #n

  .
  .
  .

Par #1

No. of parameter bytes

RA (calling routine)
                < (S)
```

STANDARD BYTE ROUTINES

For standard byte routines, parameter values are passed using specific registers. The operands are 8-bit signed or unsigned bytes. There are two groups of byte operations: the unary byte operation, and the binary byte operation. These operations are discussed in the following paragraphs.

Unary Byte Operations

Zbyteabs	Byte absolute value
Zbyteneg	Byte negation

Register Allocation Summary :: Unary byte operations

Input: B contains value to be operated on

Output: B contains the result byte

Registers:
Modified: B
Unchanged: A,X,Y,U,S,CC

Binary Byte Operations

Zbyteadd	Byte addition
Zubyteadd	Unsigned byte addition
Zbytesub	Byte subtraction
Zubytesub	Unsigned byte subtraction
Zbytemul	Byte multiplication
Zbytediv	Byte division
Zubytediv	Unsigned byte division

a. Zbyteadd performs the addition of two bytes.

b. Zbytediv performs the division of two bytes using the following algorithm:

(1) Shift divisor left to its highest possible value.

(2) Subtract divisor from dividend.

(3) If result is positive, put 1 in rightmost digit of quotient. If negative, add divisor back into dividend and put 0 in quotient.

(4) Shift divisor right and repeat steps 2 and 3 until divisor returns to its original value. The result of the division is available in register B upon completion. The remainder is also available in register A (used for MODULUS).

c. Zbytemul performs the multiplication of two bytes. The actual multiplication works on positive values and produces a positive dummy result. The routine handles negative operands by counting and complementing the negative operands using a counter which is set to -1. If one negative operand exists, the counter equals zero and causes the negation of the dummy result to obtain the correct result. If both operands are positive or negative, the positive dummy result is the actual result. The eight LSB of the result of the multiplication are available in register B upon completion of the routine and the eight MSB are in register A.

d. Zbytesub performs the subtraction of two bytes.

e. Zubyteadd performs the addition of two unsigned bytes.

f. Zubytediv performs the division of two unsigned bytes. The binary division algorithm is as that of b above. The result of the division is available in register B upon completion. The remainder is also available in register A (used for MODULUS).

g. Zubytesub performs the subtraction of two unsigned bytes.

Register Allocation Summary :: Binary 8BIT ops.

Input: B contains the first operand A contains the second operand

Output: B contains the result A contains the MSB of result - MUL contains the remainder - DIV

Registers: Modified: A,B Unchanged: X,Y,U,S,CC

STANDARD INTEGER ROUTINES

The integer operations require 16-bit operands. The two 8-bit accumulators are normally used as a 16-bit register (called D) for these routines. As a register pair, the high-order byte is always stored in register A and the low-order byte is stored in register B. Register X is a 16-bit register and is used for binary operations and for returning some results. There are two groups of integer operations: the unary integer operation and the binary integer operation. These operations are discussed in the following paragraphs.

Unary Integer Operations

Zintabs	Integer absolute value
Zintneg	Integer negation

Register Allocation Summary :: Unary integer operations

Input: D contains integer value to be operated on

Output: D contains integer RESULT

Registers:
Modified: D
Unchanged: X,Y,U,S,CC

Binary Integer Operations

Zintadd	Integer addition
Zuintadd	Unsigned integer addition
Zintsub	Integer subtraction
Zuintsub	Unsigned integer subtraction
Zintmul	Integer multiplication
Zuintmul	Unsigned integer multiplication
Zintdiv	Integer division
Zuintdiv	Unsigned integer division

a. Zintadd performs the addition of two integers.

b. Zintdiv performs the division of two integers. The actual division works on positive values and produces a positive dummy result. The routine handles negative operands by counting and complementing the negative operands using a counter which is set to -1. If one negative operand exists, the counter equals zero and causes the negation of the dummy result. If both operands are positive or negative, the dummy result is the correct answer. The division algorithm is as follows:

(1) Shift divisor left to its highest possible value.

(2) Subtract divisor from dividend.

(3) If result is positive, put 1 in rightmost digit of quotient. If negative, recover divisor before subtraction and put 0 in quotient.

(4) Shift divisor right and repeat steps b and c until divisor returns to its original value.

The result of the division is available in register D upon completion. The remainder is available in register X (used for MODULUS).

c. Zintmul performs the multiplication of two integers. The actual multiplication works on positive values and produces a positive dummy result. The routine handles negative operands by counting and complementing the negative operands using a counter which is set to -1. If one negative operand exists, the counter equals zero and causes the negation of the dummy result. If both operands are positive or negative, the positive dummy result is the correct result. The multiplication occurs as follows:

(A:B) * (C:D) = BDH : BDL + BCH: BCL + ADH: ADL + ACH: ACL

The lower 16 bits of the result are placed into the D register and the 16 most significant bits of the result are placed in register X upon completion of the library routine.

d. Zintsub performs the subtraction of two integers.

e. Zuintadd performs the addition of two unsigned integers.

f. Zuintdiv performs the division of two unsigned integers. The binary division algorithm is as that in b above. The result of the division is available in register D upon completion. The remainder is available in register X (for MOD).

g. Zuintmul performs the multiplication of two unsigned integers. The actual multiplication occurs as explained in c above. The lower 16 bits of the result are placed into register D and the 16 most significant bits of the result are placed in register X upon completion of the library routine.

h. Zuintsub performs the subtraction of two unsigned integers.

Register Allocation Summary :: Binary 16BIT ops.

Input: X contains the first operand D contains the second operand Output: D contains the result X contains the MSW of result - MUL contains the remainder - DIV

Registers: Modified: D,X Unchanged: Y,U,S,CC

BYTE AND WORD SHIFTS

C/64000 supports logical shifting. Circular shifting may be accomplished by calling the run-time library ROTATE. The DIV operator using powers of 2 may be used to accomplish an arithmetic right shift (i.e., with sign extension). For example, X DIV 2 is equivalent to a one bit right shift with sign extension.

SHIFT

Logical shifting with zero fill will shift the quantity left or right placing a zero in the most (right shift) or least (left shift) significant bit for each shift. The function is called with two parameters: the quantity to be shifted and the number of bit positions to shift.

ROTATE

Circular shifting rotates the quantity left or right and fills the vacated position with the bit shifted out of the other end. The function is called with two parameters: the quantity to be shifted and the number of bit positions to shift.

The type of result returned by the function SHIFT or ROTATE is the same type as the type of the first parameter expression.

Byte Shifts

```
Zbshift      Byte shift logical with zero fill
Zbrotate     Byte shift circular
```

The byte shift operations compute the byte result of shift expressions.

Register Allocation Summary :: Byte shift operations

Input: A contains byte to be shifted, B1
B contains number of positions to shift, B2

Output: B contains the result byte

Registers:
Modified: B,CC
Unchanged: A,X,Y,U,S,DP

Word Shifts

```
Zwshift      Word shift logical with zero fill
Zwrotate     Word shift circular
```

Register Allocation Summary :: Integer shift operations

Input: X contains word to be shifted, I1
B contains the number of positions to shift, I2

Output: D contains the result word

Registers:
Modified: D
Unchanged: X,Y,U,S,DP

BYTE AND WORD SET OPERATIONS

Byte Set Operations

```
Zbinset8     Byte in 8-bit set
Zbtoset8     Byte to 8-bit set
```

Zbinset8. This routine is used to test the set membership of a byte value in a specified byte set.

Register Allocation Summary :: Zbinset8

Input: B contains the byte set being compared
A contains byte value to be tested

Output: B set to 0, Z flag set if value not in set
B set to 1, Z flag reset if value in set

Registers:
Modified: B,CC
Unchanged: A,X,Y,U,S,DP

At termination, register A will contain the byteset which was compared.

Zbtoset8. This routine converts a byte into an 8-bit set. The only valid input values are 0 through 7. Out of range values are detected in the debug library, DLIB_6809:C6809, but are not detected in LIB_6809:C6809 and may produce out of range results.

Register Allocation Summary :: Zbtoset8

Input: B contains byte value to be converted

Output: B contains the byteset result

Registers:
Modified: B
Unchanged: A,X,Y,U,S,CC

Word Set Operations

```
Zbinset16     Byte in 16-bit set
Zbtoset16     Byte to 16-bit set
Zwinset16     Word in 16-bit set
Zwtoset16     Word to 16-bit set
```

Zbinset16. This routine is used to test the set membership of a byte value in a specified word set.

Register Allocation Summary :: Zbinset16

Input: B contains byte value to be tested
X contains the word set being compared

Output: B set to 0, Z flag set if value not in set
B set to 1, Z flag reset if value in set

Registers:
Modified: B,CC
Unchanged: A,X,Y,U,S,DP

At termination, register X will contain the word set compared.

Zbtoset16. This routine converts a byte into a 16-bit set. The only valid input values are 0 through 15. Out of range values are detected in the debug library, DLIB_6809:C6809, but are not detected in LIB_6809:C6809 and may produce out of range results.

Register Allocation Summary :: Zbtoset16

Input: B contains byte value to be converted

Output: D contains the wordset result

Registers:
Modified: D
Unchanged: X,Y,U,S,CC

Zwinset16. This routine is used to test the set membership of a word value in a specified word set.

Register Allocation Summary :: Zwinset16

Input: D contains word value to be tested
X contains the word set being compared

Output: B set to 0, Z flag set if value not in set
B set to 1, Z flag reset if value in set

Registers:
Modified: D,CC
Unchanged: X,Y,U,S,DP

Zwtoset16. This routine converts a word into a 16-bit set. The only valid input values are 0 through 15. Out of range values are detected in the debug library, DLIB_6809:C6809, but are not detected in LIB_6809:C6809 and may produce out of range results.

Register Allocation Summary :: Zwtoset16

Input: D contains word value to be converted

Output: D contains the wordset result

Registers:
Modified: D
Unchanged: X,Y,U,S,CC

MULTI-BYTE OPERATIONS

The multi-byte routines are used by the compiler to operate on multi-byte records (or arrays) of the same type.

MBmove

The routine MBmove is used for moving multi_byte records such as in an assignment of a complete record type or an array type to another of the same type.

Register Allocation Summary :: MBmove

Input: X contains the first record's addr.
U contains the second records's addr.
D contains the number of bytes in
the records.

Registers:
Modified : D
Unchanged: X,Y,U,S,CC

Multi-byte Comparisons

```
MBequ          Multi-byte equality test
MBneq          Multi-byte inequality test
MBgeq          Multi-byte greater than or equal test
MBgtr          Multi-byte greater than test
MBleq          Multi-byte less than or equal test
MBles          Multi-byte less than test
```

MBequ. This routine is used by the compiler to test multi-byte records of the same type for equality.

MBneq. This routine is used by the compiler to test multi-byte records of the same type for inequality.

MBgeq. This routine is used by the compiler to test if one set of records is greater than or equal to another set of records of the same type. The test is unsigned.

MBgtr. This routine is used by the compiler to test if one set of multi-byte records is greater than another set of the same type. The test is unsigned.

MBleq. This routine is used by the compiler to test if one set of records is less than or equal to another set of records of the same type. The test is unsigned.

MBles. This routine is used by the compiler to test if one set of records is less than another set of records of the same type. The test is unsigned.

A compare routine is called to compare the bytes and upon re-entry to this program, a branch is taken to either a "true" or "false" routine.

Register Allocation Summary :: Multi-bytes

Input: X contains the first record's addr.
U contains the second records's addr.
D contains the number of bytes in the records.

Registers:
Modified : D
Unchanged: X,Y,U,S,CC

Output:

test results	B register	Z flag
true	1	reset
false	0	set

Additionally, register A will contain the byte within the first set of bytes which caused the equality comparison to fail or was the last equal byte to be compared.

MULTI-BYTE SET OPERATIONS

Sets requiring three or more bytes, are referred to as multi-byte sets.

Multi-byte Set Routines

INSETmb	Multi-byte set inclusion
TOSETmb	Multi-byte set formation
SETmbINT	Multi-byte set intersection
SETmbUNI	Multi-byte set union
SETmbDIF	Multi-byte set difference
SETmbLEQ	Multi-byte subset inclusion

INSETmb. This routine is used to test the set membership of an integer value in a multi-byte set.

```
Register Allocation Summary :: INSETmb

Input : X contains address of the multi-byte
        set
       D contains the integer value V

Output: IF V is contained in set
        THEN
          B set to 1 (TRUE), Z flag reset
        ELSE
          B set to 0 (FALSE), Z flag set

Registers:
   Modified : D,CC
   Unchanged: X,Y,U,S,DP
```

TOSETmb. This routine is used to convert a value into a multi-byte set.

```
Register Allocation Summary :: TOSETmb

Input:  X contains byte value to be converted
        U contains the addr. of the result set
        D contains the number of bytes in the
          set

Registers:
   Modified:  D
   Unchanged: X,Y,U,S,CC
```

SETmbINT. This routine is used to compute the set intersection of two multi-byte sets.

SETmbUNI. This routine is used to compute the set union of two multi-byte sets.

SETmbDIF. This routine is used to compute the set difference of two multi-byte sets. The set difference is a set containing all the elements of the multi-byte set in (X) which are not contained in the multi-byte set in (U).

Register Allocation Summary :: Big sets

Input : X contains the first set's adr.
U contains the second set's adr.
Y contains the result set's adr.
D contains the number of bytes

Registers:
Modified : D
Unchanged: X,Y,U,S,CC

SETmbLEQ. This routine is used to compute the set inclusion of two multi-byte sets.

Register Allocation Summary :: SETmbLEQ

Input : X contains address of S1
U contains address of S2
D contains the number of bytes

Output: IF S2 is a subset of S1
THEN
B set to 1 (TRUE), Z flag reset
ELSE
B set to 0 (FALSE), Z flag set

Registers:
Modified : D,CC
Unchanged: X,Y,U,S

To accomplish the "proper subset" operation, the 6809 compiler will invert the operands and proceed to call SETmbLEQ.

BYTE AND INTEGER COMPARISON AND BOUNDS

Checking Routines

The comparison (=,<>,>=,>,<=,<) of byte and integer variables produces a Boolean result (FALSE or TRUE) based on the signed or unsigned sequences of byte or word scalar types. In many cases where the comparison is being used as the condition for an IF, REPEAT, or WHILE statement, a branch is taken based on the result of the comparison. However, if the Boolean result is being assigned to a variable or if the expression has multiple comparisons (using AND and OR) an actual Boolean result is required. The byte and word comparison subroutines are used specifically in these situations were the Boolean result is necessary for further computations.

Byte and Word Comparisons

Zcc	Carry clear test
Zequ	Byte and integer equality test
Zneq	Byte and integer inequality test
Zgeq	Byte and integer greater than or equal test
Zgtr	Byte and integer greater than test
Zleq	Byte and integer less than or equal test
Zles	Byte and integer less than test
Zugeq	Unsigned byte and integer greater than or equal test
Zugtr	Unsigned byte and integer greater than test
Zuleq	Unsigned byte and integer less than or equal test
Zules	Unsigned byte and integer less than test

Library subroutines are called when the Boolean result is required of a comparison expression of the following form: I1 .op. I2

Zcc is called to test if the carry bit is cleared.

Zequ is called to test for equality between I1 and I2 by testing if the Z flag of the condition codes is set.

Zneq is called to test for inequality between I1 and I2 by testing if the Z bit of the condition codes is set.

Zgeq is called to test if I1 is greater than or equal to I2 by testing if either, but not both, of the N or V bits of the condition codes is set.

Zgtr is called to test if I1 is greater than I2 by testing if the "EXCLUSIVE OR " of the N and V bits is 1 or Z=1.

Zleq is called to test if I1 is less than or equal to I2 by testing if the "EXCLUSIVE OR" of the N and V bits is 1 or Z=1.

Zles is called to test if either, but not both, of the N or V bits is set.

Zugeq is called to test if I1 is less than I2 by testing if the C flag of the condition codes is set.

Zugtr is called to test if I1 is greater than I2 by testing if the previous operation caused either a carry or a zero result.

Zuleq is called to test if I1 is less than or equal to I2 by testing if the previous operation caused either a carry or a zero result.

Zules is called to test if the C bit is set or not.

Output:

test results	B register	Z flag
true	1	reset
false	0	set

Byte Bounds Checking

```
Zbbounds     Byte bounds checking
Zubbounds    Unsigned byte bounds checking
```

The bounds checking for!signed and unsigned byte variables use the same calling sequence and return the same results. The value being checked is loaded into register X. The upper limit is loaded into register A and the lower limit is loaded into register B. Upon return, register B contains the Boolean result (FALSE or TRUE) of the bounds check and the Z flag will be set according to the Boolean value in B. If B=FALSE (0) then Z is set. If B=TRUE (1) then Z is reset.

```
Register Allocation Summary :: Byte bounds check

Input:  B         contains the value V
        MSB of X  contains the lower limit (LL)
        LSB of X  contains the upper limit (UL)

Output: B set to 0, Z flag set if value not in range
        B set to 1, Z flag reset if value in range

Registers:
    Modified:  B,CC
    Unchanged: X,Y,U,S
```

Word Bounds Checking

```
Zwbounds     Integer bounds checking
Zuwbounds    Unsigned integer bounds checking
```

The bounds checking for signed and unsigned word variables use the same calling sequence and return the same results. The value being checked is loaded into register X. The upper limit is loaded into register D and the lower limit is loaded into register U. Upon return, register B contains the Boolean results (FALSE or TRUE) of the bounds check and the Z flag will be set according to the Boolean value of register B. If B=FALSE then Z is set. If B=TRUE than Z is reset.

The logic of the routine is:

```
IF  UL <= V <= LL     THEN true_result
                      ELSE false_result

Register Allocation Summary :: Word bounds check

Input:  D contains the upper limit (UL)
        U contains the lower limit (LL)
        X contains the value V

Output: B set to 0, Z flag set if value not in range
        B set to 1, Z flag reset if value in range

Registers:
    Modified:  B,CC
    Unchanged: X,Y,U,S
```

STRINGS AND CHARACTERS

The routines STequ, STneq, STgeq, STgtr, STleq, STles are used by the compiler to test strings equality or inequality.

The routines CHequ, CHneq, CHgeq, CHgtr, CHleq, CHles are used by the compiler to test strings .vs. characters equality or inequality. The character is always the first argument (the compiler will invert the relational operand if necessary, i.e. ST <= CH becomes CH > ST). This routines will set up their arguments and then call the string routines.

A compare routine is called to compare the bytes and upon re-entry to this program, a branch is taken to either a "true" or "false" routine.

String equality and inequality in the C compiler are determined by the following rules:

a. Two strings are equal IF their lengths are equal and they are equal character by character.

b. The inequality of two strings is determined by the first character by which they differ and if all characters are equal then the longest string is the largest.

Register Allocation Summary: String routines

Input : X contains the first string's addr.
U contains the second string's addr.
D contains the number of bytes in
the string's type

Registers:
Modified : D
Unchanged: X,Y,U,S,CC

Register Allocation Summary: Character routines

Input : B contains the character (first operand)
U contains the string's address
D contains the number of bytes in the
string's type

Registers:
Modified : D
Unchanged: X,Y,U,S,CC

Output:

test results	B register	Z flag
true	1	reset
false	0	set

Additionally, register A will contain the byte within the first set of bytes which caused the equality comparison to fail or was the last equal byte to be compared.

STmove

The routine STmove is used to copy a string from one location to another.

Register Allocation Summary :: STmove

Input: X contains the first string's addr.
U contains the second string's addr.
D contains the number of bytes in the
first string's type (n1+1)

Registers:
Modified : D
Unchanged: X,Y,U,S,CC

Chapter 4
RUN-TIME LIBRARY
SPECIFICATIONS FOR REAL NUMBERS

REAL NUMBER LIBRARIES

The C/64000 implementation of the IEEE floating point standard for the 6809 microprocessor is supported by the real library: RealLIB:C6809 (for C data types: "double" and "float"). Table 4-1 summarizes the standard C floating point routines. These routines use the parameter passing conventions described for normal C static functions with the option $FIXED_PARAMETERS OFF$ ($FIXED_PARAMETERS OFF$ is the default; see Chapter 2 "Options").

Table 4-1. C Real Number Library Routines ($FIXED_PARAMETERS OFF$)

Name	Purpose	Ref Page
ABS	Double absolute value	4-3
SQRT	Double square root	4-3
EXP	Double exponentiation(e to the X)	4-3
LN	Double natural logarithm	4-3
SIN	Double sine	4-3
COS	Double cosine	4-3
ARCTAN	Double arctangent	4-3

Table 4-2 summarizes the rest of the floating point routines supported by the C/64000 real number library. These routines use the parameter passing method obtained with the compiler option $FIXED_PARAMETERS ON$.

The parameter passing for these routines is that described in "User Defined Operators" (see Chapter 2). Each library routine has an external interface using a global symbol in the form: REAL_op or LONGREAL_op, where op is the mnemonic for one of the supported operations. Since the compiler performs some automatic type conversions there are some additional operations to convert between "int", "float" and "double" data types. Each of the library routines is defined by the equivalent C procedure heading for its declaration.

Table 4-2. C Real Number Library Routines ($FIXED_PARAMETERS ON$)

Name	Purpose	Ref Page
REAL_ADD	Real addition	4-4
REAL_SUB	Real subtraction	4-4
REAL_MUL	Real multiplication	4-4
REAL_DIV	Real division	4-4
REAL_ABS	Real absolute value	4-5
REAL_NEG	Real negation	4-5
REAL_SQRT	Real square root	4-5
REAL_EXP	Real exponentiation(e to the X)	4-5
REAL_LN	Real natural logarithm	4-5
REAL_SIN	Real sine	4-5
REAL_COS	Real cosine	4-5
REAL_ATAN	Real arctangent	4-5
REAL_EQU	Real equality test	4-6
REAL_NEQ	Real inequality test	4-6
REAL_LES	Real less than test	4-6
REAL_GTR	Real greater than test	4-6
REAL_LEQ	Real less than or equal test	4-6
REAL_GEQ	Real greater than or equal test	4-6
REAL_FLOAT	Integer to real conversion	4-7
REAL_ROUND	Real to integer conversion with rounding	4-7
REAL_TRUNC	Real to integer conversion with truncation	4-7
LONGREAL_ADD	Longreal addition	4-4
LONGREAL_SUB	Longreal subtraction	4-4
LONGREAL_MUL	Longreal multiplication	4-4
LONGREAL_DIV	Longreal division	4-4
LONGREAL_ABS	Longreal absolute value	4-5
LONGREAL_NEG	Longreal negation	4-5
LONGREAL_SQRT	Longreal square root	4-5
LONGREAL_EXP	Longreal exponentiation(e to the X)	4-5
LONGREAL_LN	Longreal natural logarithm	4-5
LONGREAL_SIN	Longreal sine	4-5
LONGREAL_COS	Longreal cosine	4-5
LONGREAL_ATAN	Longreal arctangent	4-5
LONGREAL_EQU	Longreal equality test	4-6
LONGREAL_NEQ	Longreal inequality test	4-6
LONGREAL_LES	Longreal less than test	4-6
LONGREAL_GTR	Longreal greater than test	4-6
LONGREAL_LEQ	Longreal less than or equal test	4-6
LONGREAL_GEQ	Longreal greater than or equal test	4-6
LONGREAL_FLOAT	Integer to longreal conversion	4-7
LONGREAL_ROUND	Longreal to integer conversion with rounding	4-7
LONGREAL_TRUNC	Longreal to integer conversion with trucation	4-7
REAL_CONTRACT	Longreal to real conversion	4-7
REAL_EXTEND	Real to longreal conversion	4-7

C REAL NUMBER LIBRARY ROUTINES

($FIXED_PARAMETERS OFF$)

NOTE

The routines in this section, use the parameter passing method that is generated with the default option $FIXED_PARAMETERS OFF$.

The unary functions using the standard C compatible method for passing parameters ($FIXED_PARAMETERS OFF$) supported in the library, RealLIB:C6809, are as follows:

```
double ABS  ( ARGUMENT )    /* double absolute value */
   double ARGUMENT;
double NEG  ( ARGUMENT )    /* double negation */
   double ARGUMENT;
double SQRT ( ARGUMENT )    /* double square root */
   double ARGUMENT;
double EXP  ( ARGUMENT )    /* double exponentiation */
   double ARGUMENT;
double LN   ( ARGUMENT )    /* double natural logarithm */
   double ARGUMENT;
double SIN  ( ARGUMENT )    /* double sine(radians) */
   double ARGUMENT;
double COS  ( ARGUMENT )    /* double cosine(radians) */
   double ARGUMENT;
double ARCTAN ( ARGUMENT )  /* double arctangent */
   double ARGUMENT;
```

C REAL NUMBER LIBRARY ROUTINES

($FIXED_PARAMETERS ON$)

NOTE

The routines in this section use the parameter passing method generated with the option $FIXED_PARAMETERS ON$.

FLOATING POINT BINARY OPERATIONS

For binary floating point operations of the form:

RESULT = LEFT <op> RIGHT;

the equivalent C procedure heading is in the form:

REAL_<op> (LEFT,RIGHT,RESULT) float *LEFT,*RIGHT,*RESULT; or LONGREAL_<op> (LEFT,RIGHT,RESULT) double *LEFT,*RIGHT,*RESULT;

The binary operations supported in RealLIB:C6809 are as follows:

```
REAL_ADD ( LEFT,RIGHT,RESULT )    /* float addition */
   float *LEFT,*RIGHT,*RESULT;
REAL_SUB ( LEFT,RIGHT,RESULT )    /* float subtraction */
   float *LEFT,*RIGHT,*RESULT;
REAL_MUL ( LEFT,RIGHT,RESULT )    /* float multiplication
*/
   float *LEFT,*RIGHT,*RESULT;
REAL_DIV ( LEFT,RIGHT,RESULT )    /* float division */
   float *LEFT,*RIGHT,*RESULT;
LONGREAL_ADD ( LEFT,RIGHT,RESULT )/* double addition */
   double *LEFT,*RIGHT,*RESULT;
LONGREAL_SUB ( LEFT,RIGHT,RESULT )/* double subtraction */
   double *LEFT,*RIGHT,*RESULT;
LONGREAL_MUL ( LEFT,RIGHT,RESULT )/* double multiplication
*/
   double *LEFT,*RIGHT,*RESULT;
LONGREAL_DIV ( LEFT,RIGHT,RESULT )/* double division */
   double *LEFT,*RIGHT,*RESULT;
```

FLOATING POINT UNARY OPERATIONS

For unary floating point operations of the form:

RESULT = <op> RIGHT;

the equivalent C procedure heading is in the form:

REAL_<op> (RIGHT,RESULT) float *RIGHT,*RESULT; or LONGREAL_<op> (RIGHT,RESULT) double *RIGHT,*RESULT;

The unary operations supported in RealLIB:C6809 are as follows:

```
REAL_ABS  ( RIGHT,RESULT )        /* float absolute value
          */
   float *RIGHT,*RESULT;
REAL_NEG  ( RIGHT,RESULT )        /* float negation */
   float *RIGHT,*RESULT;
REAL_SQRT ( RIGHT,RESULT )        /* float square root */
   float *RIGHT,*RESULT;
REAL_EXP  ( RIGHT,RESULT )        /* float exponentiation
          */
   float *RIGHT,*RESULT;
REAL_LN   ( RIGHT,RESULT )        /* float natural
          logarithm*/
   float *RIGHT,*RESULT;
REAL_SIN  ( RIGHT,RESULT )        /* float sine */
   float *RIGHT,*RESULT;
REAL_COS  ( RIGHT,RESULT )        /* float cosine */
   float *RIGHT,*RESULT;
REAL_ATAN ( RIGHT,RESULT )        /* float arctangent */
   float *RIGHT,*RESULT;
LONGREAL_ABS  ( RIGHT,RESULT )    /* double absolute value
              */
   double *RIGHT,*RESULT;
LONGREAL_NEG  ( RIGHT,RESULT )    /* double negation */
   double *RIGHT,*RESULT;
LONGREAL_SQRT ( RIGHT,RESULT )    /* double square root */
   double *RIGHT,*RESULT;
LONGREAL_EXP  ( RIGHT,RESULT )    /* double exponentiation
              */
   double *RIGHT,*RESULT;
LONGREAL_LN   ( RIGHT,RESULT )    /* double natural
              logarithm */
   double *RIGHT,*RESULT;
LONGREAL_SIN  ( RIGHT,RESULT )    /* double sine */
   double *RIGHT,*RESULT;
LONGREAL_COS  ( RIGHT,RESULT )    /* double cosine */
   double *RIGHT,*RESULT;
LONGREAL_ATAN ( RIGHT,RESULT )    /* double arctangent */
   double *RIGHT,*RESULT;
```

FLOATING POINT COMPARISON OPERATIONS

For floating point comparison operations of the form:

BOOLEAN = LEFT <op> RIGHT;

the equivalent C procedure heading is in the form:

short REAL_<op> (LEFT,RIGHT) float *LEFT,*RIGHT; or short LONGREAL_<op> (LEFT,RIGHT) double *LEFT,*RIGHT;

The comparison operations supported in RealLIB:C6809 are as follows:

```
short REAL_EQU ( LEFT,RIGHT )      /* float equality test */
    float *LEFT,*RIGHT;
short REAL_NEQ ( LEFT,RIGHT )      /* float inequality test
*/
    float *LEFT,*RIGHT;
short REAL_LES ( LEFT,RIGHT )      /* float less than test
*/
    float *LEFT,*RIGHT;
short REAL_GTR ( LEFT,RIGHT )      /* float greater than
test */
    float *LEFT,*RIGHT;
short REAL_LEQ ( LEFT,RIGHT )      /* float less than or
equal test */
    float *LEFT,*RIGHT;
short REAL_GEQ ( LEFT,RIGHT )      /* float greater than or
equal test */
    float *LEFT,*RIGHT;
short LONGREAL_EQU ( LEFT,RIGHT )  /* double equality test
*/
    double *LEFT,*RIGHT;
short LONGREAL_NEQ ( LEFT,RIGHT )  /* double inequality test
*/
    double *LEFT,*RIGHT;
short LONGREAL_LES ( LEFT,RIGHT )  /* double less than test
*/
    double *LEFT,*RIGHT;
short LONGREAL_GTR ( LEFT,RIGHT )  /* double greater than
test */
    double *LEFT,*RIGHT;
short LONGREAL_LEQ ( LEFT,RIGHT )  /* double less than or
equal test */
    double *LEFT,*RIGHT;
short LONGREAL_GEQ ( LEFT,RIGHT )  /* double greater than or
equal test */
    double *LEFT,*RIGHT;
```

FLOATING POINT CONVERSION OPERATIONS

For floating point conversion operations of the form:

RESULT = <op> RIGHT;

the equivalent C procedure heading is in the form:

REAL_<op> (RIGHT, RESULT) RIGHTtype *RIGHT; RESULTtype *RESULT; or
LONGREAL_<op> (RIGHT, RESULT) RIGHTtype *RIGHT; RESULTtype *RESULT;

The conversion operations supported in RealLIB:C6809 are as follows:

```
REAL_FLOAT  ( RIGHT, RESULT )      /* convert int to float
            */
   int *RIGHT;float *RESULT;
REAL_ROUND  ( RIGHT, RESULT )      /* convert float to int
            */
   float *RIGHT; int *RESULT;         /* with rounding */
REAL_TRUNC  ( RIGHT, RESULT )      /* convert float to int
            */
   float *RIGHT; int *RESULT;         /* with trunctation
   */
LONGREAL_FLOAT ( RIGHT,  RESULT )  /* convert int to double
*/
   int *RIGHT; double *RESULT;
LONGREAL_ROUND ( RIGHT, RESULT )   /* convert double to int
*/
   double *RIGHT; int *RESULT;        /* with rounding */
LONGREAL_TRUNC ( RIGHT, RESULT )   /* convert double to int
*/
   double *RIGHT; int *RESULT;        /* with truncation */
REAL_CONTRACT  ( RIGHT, RESULT )   /* convert double to
               float */
   double *RIGHT; float *RESULT;
REAL_EXTEND    ( RIGHT, RESULT )   /* convert float to
               double */
   float *RIGHT; double *RESULT;
```

FLOATING POINT ERROR DETECTION

The floating point libraries have two error conditions which when detected cause the execution of one of two global routines. These routine names are OVERFLOW and INVALID. OVERFLOW is called when an operation would produce an invalid number. INVALID is called when an invalid floating point number is passed as a parameter to one of the floating point routines.

The user may replace either of these routines with an error recovery routine of his own. In particular defining either of these routines as a simple return from subroutine instruction (RTS) will cause the program to continue with an invalid number returned as a result. If OVERFLOW or INVALID are to be defined in a C program, they must have no parameters with the option $FIXED_PARAMETERS ON$.

If the user does not supply his or her own version of these routines, the libraries will supply one which will display a message on the buffer ERROR_MESSAGE, and then return to the library routine.

If either of the illegal opcodes is detected by the emulator, the user can get information describing the error by entering the emulation command:

display memory ERROR_MESSAGE blocked word

which will produce a memory display indicating the error condition.

If no error has occurred, the display will appear as follows:

```
Memory    :words  :blocked :repetitively
address        data              :hex                  :ascii
-------------------------------------------------------------------

9003-12   4E6F 2065 7272 6F72 2020 2020 2020 2020   No error
9013-22   2020 2020 2020 2020 2020 2020 2020 2020
9023-32   2020 2020 2020 2020 2020 2020 2020 2020
9033-42   2020 2020 2020 2020 2020 2020 2020 2020
9043-52   2020 2020 2020 2020 2020 2020 2020 2020
9053-62   2020 2020 2020 2020 2020 2020 2020 2020
9063-72   2020 2020 2020 2020 2020 2020 2020 2020
9073-82   2020 2020 2020 2020 2020 2020 2020 2020
9083-92   2020 2020 2020 2020 2020 2020 2020 2020
9093-A2   2020 2020 2020 2020 2020 2020 2020 2020
90A3-B2   2020 2020 2020 2020 2020 2020 2020 2020
90B3-C2   2020 2020 2020 2020 2020 2020 2020 2020
90C3-D2   2020 2020 2020 2020 2020 2020 2020 2020
90D3-E2   2020 2020 2020 2020 2020 2020 2020 2020
90E3-F2   2020 2020 2020 2020 2020 2020 2020 2020
90F3-02   2020 2020 2020 2020 2020 2020 2020 2020
```

```
Memory    :words  :blocked :repetitively
address         data              :hex                  :ascii
-------------------------------------------------------------------

9003-12   5265 616C 2020 2020 6572 726F 7220 2020  Real    error
9013-22   494E 5641 4C49 4420 2020 2020 2020 2020  INVALID
9023-32   5245 414C 5F41 4444 2020 2020 2020 2020  REAL_ADD
9033-42   726F 7574 696E 6520 6361 6C6C 6564 2020  routine called
9043-52   6279 2020 2020 2020 7573 6572 2020 2020  by  user
9053-62   6672 6F6D 2020 2020 6164 6472 6573 7320  from   address
9063-72   3143 3137 482E 2020 2020 2020 2020 2020  1C17H.
9073-82   2020 2020 2020 2020 2020 2020 2020 2020
9083-92   2020 2020 2020 2020 2020 2020 2020 2020
9093-A2   2020 2020 2020 2020 2020 2020 2020 2020
90A3-B2   2020 2020 2020 2020 2020 2020 2020 2020
90B3-C2   2020 2020 2020 2020 2020 2020 2020 2020
90C3-D2   2020 2020 2020 2020 2020 2020 2020 2020
90D3-E2   2020 2020 2020 2020 2020 2020 2020 2020
90E3-F2   2020 2020 2020 2020 2020 2020 2020 2020
90F3-02   2020 2020 2020 2020 2020 2020 2020 2020
```

```
Memory    :words  :blocked :repetitively
address         data              :hex                  :ascii
-------------------------------------------------------------------

9003-12   5265 616C 2020 2020 6572 726F 7220 2020  Real    error
9013-22   4F56 4552 464C 4F57 2020 2020 2020 2020  OVERFLOW
9023-32   4C4F 4E47 5245 414C 5F41 4444 2020 2020  LONGREAL _ADD
9033-42   726F 7574 696E 6520 6361 6C6C 6564 2020  routine called
9043-52   6279 2020 2020 2020 7573 6572 2020 2020  by    user
9063-72   3144 3535 482E 2020 2020 2020 2020 2020  from   address
9073-82   2020 2020 2020 2020 2020 2020 2020 2020  1D55H.
9083-92   2020 2020 2020 2020 2020 2020 2020 2020
9093-A2   2020 2020 2020 2020 2020 2020 2020 2020
90A3-B2   2020 2020 2020 2020 2020 2020 2020 2020
90B3-C2   2020 2020 2020 2020 2020 2020 2020 2020
90C3-D2   2020 2020 2020 2020 2020 2020 2020 2020
90D3-E2   2020 2020 2020 2020 2020 2020 2020 2020
90E3-F2   2020 2020 2020 2020 2020 2020 2020 2020
90F3-02   2020 2020 2020 2020 2020 2020 2020 2020
```

From this display the user can tell what type error has been detected, which floating point library was called, and where the floating point library detected the error.

NOTES

Appendix A
RUN-TIME ERROR DESCRIPTIONS

This appendix contains descriptions of run-time errors that may occur.

ERROR UTILITIES

Name	Purpose
Derrors	Debugging library error handler
Zerrors	Normal library error handler

Derrors

Derrors contains the run-time routines which store user information at the time an error occurs during debugging. The following errors may occur in the indicated library routines:

Error	Routines
Underflow	Zbytemul,Zintmul,Zuintmul Zbyteadd, Zubyteadd, Zintadd, Zuintadd Zbytesub, Zubytesub, Zintsub, Zuintsub
Overflow	Zbytemul, Zintmul, Zuintmul Zbytediv, Zubytediv, Zintdiv, Zuintdiv Zbyteadd, Zubyteadd, Zintadd, Zuintadd Zbytesub, Zubytesub, Zintsub, Zuintsub Zbyteneg, Zintneg Zbyteabs, Zintabs

Div_by_Zero	Zbytediv, Zubytediv, Zintdiv, Zuintdiv
Case_error	User programs
Range_error	COMPB_ .
Heap_error	INITHEAP, NEW, DISPOSE, MARK, RELEASE
Set_conversion_error	Zbtoset8, Zwtoset8 Zbtoset16, Zwtoset16
String_error	MOVEST_ .

When an error is detected, a jump to Derrors is generated and valid register information is saved. The labels for the stored information are described below:

Label	Description
Z_CALLER_H Z_CALLER_L	Contain the high byte (CALLER_H) and the low byte (CALLER_L) of the address of the statement which called the routine where the actual error occurred.
Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X Z_REG_U	Contain the contents of the registers at the time the error occurred. Only registers with information relevant to the error are saved - the indicated contents of the other registers is garbage.

NOTE

The CC register which is displayed is that which was present when the error occurred in the Debug Library routine. The CC register which was present when the Debug routine was called is not retrievable.

The following is a description of the errors that may occur and the information that is accessible when they do occur.

Error Msg. Available	Description	Information
Z_ERR_OVERFLOW	Jump to error occurs when results of multiplication, addition, subtraction, negation, or the absolute value is too positive (i.e. INTEGERS: result > 32767 BYTES: result > 127)	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X Z_REG_U
Z_ERR_UNDERFLOW	Jump to error occurs if results of addition, subtraction, or multiplication were too negative (i.e. INTEGERS result < -32768 BYTES result < -128)	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X Z_REG_U
Z_ERR_DIV_BY_0	Jump to error occurs if division by zero is attempted by byte or integer division routines.	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_REG_X Z_REG_U
Z_ERR_SET_CONV	Jump to error occurs if operand is not legal ordinal value for a set of the base type.	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_X

Z_ERR_RANGE	Jump to error occurs if a range declaration has been violated (i.e.: a variable does not fall within its assigned range)	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS Z_ACC_A Z_ACC_B Z_REG_U
Z_ERR_HEAP	Jump to error occurs when some misuse of the dynamic allocation keywords NEW, DISPOSE,MARK, or RELEASE takes place.	Z_CALLER_H Z_CALLER_L Z_CC_FLAGS ?
Z_ERR_CASE	Jump to error occurs when the test variable of CASE statement is out of range and no OTHERWISE label exists.	Z_CALLER_H Z_CALLER_L Z_ACC_A Z_ACC_B
Z_ERR_STRING	Jump to error occurs on a string assingment, when the run-time size of the string being assigned is larger than that of which is it is being assigned to.	Z_CALLER_H Z_CALLER_L
Z_END_PROGRAM	Jump to message occurs when the program completes execution of the main body code.	

The illegal opcodes associated with the various errors are as follows:

Opcode	Error
01	Overflow
02	Div_by_0
05	Case_error
14	Range_error
15	Recursive_error
18	Heap_error
38	Set_conversion_error
41	Underflow
42	String_size_assignment_error

Zerrors

Zerrors contains the run-time routines which store user information at the time an error occurs during execution in the non-debug library. The following errors may occur in the indicated library routines:

Error	Routines
Case_error	User programs
Range_error	COMPB_ .
Heap_error	INITHEAP, NEW, DISPOSE, MARK, RELEASE
String_error	MOVEST_ .

When an error is detected, a jump to Zerrors is generated and valid register information is saved. The stored information, the routines and the illegal opcodes for this errors are as described in Derrors.

Z_END_PROGRAM is also called.

NOTES

Index

The following index lists important terms and concepts of this manual along with the location(s) where they can be found. The numbers to the right of the listings indicate the following manual areas:

- Chapters - References to chapters appear as "Chapter X", where "X" represents the chapter number.
- Appendices - References to appendices appear as "Appendix Y" where "Y" represents the letter designator of the appendix.
- Figures/Tables - References to figures or tables are represented by the capital letter "F" or "T" followed by the appropriate number.
- Other entries in the index - Other entries in the index have their location indicated by page number.

a

b

c

d

e

f

g

i

l

m

n

o

p

r

s

u

v

w

Z

NOTES

64822-90901, MAY 1985
Replaces: 64822-90901, March 1983

PRINTED IN U.S.A.